



## Description of the application

The present apparatus includes two techniques of comparison, which when desired, may be combined to provide the full solution described above.

### Standard Compare

#### Data entry

the data needed for performing said compare operation shall include: a) product names, b) topics the merchant wishes to mention to a site visitor. These topics are defined per each product separately. C) Values of said topics for said each product.

This data is to reside in an ASCII file that will be read by the application and structured as "cmp\_data(Topic, Value, ProductName)" entries.

## Describe Structure

### Algorithms

#### General

The algorithm works as follows:

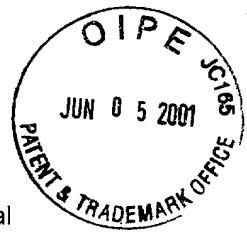
The application is launched by a request made either by another application (via pipe, for example) or by HTTP request from a web page. Provided the request will contain the names of the products needed to be compared by the application, and the topics by which the comparison is to be made, the application shall perform the following steps: 1) Retrieve the information relevant to said products. 2) Harvest the values of said topics (if specified) for said products from the information retrieved in step 1. 3) Perform the comparison of said products with the use of the respective topic-value pairs of each product and 4) construct a natural language output to report the results of the comparison. After sending said output (using a pipe, standard output write functions or the like) the application shall exit.

#### Detailed description of the algorithms

##### CompareProducts

This is the main procedure of the comparison module. This routine is to receive the request to compare some given products using one of the three comparison techniques: standard, fuzzy or intermixed. This routine is to read the data entries located in the external files to retrieve the necessary information about said products and store the information in the memory to

09829961.000001



make it accessible to the following routines. After this task is accomplished, CompareProducts will make a call to "stdCompareProducts\_ByTopics" to perform the comparison, followed by "ConstructNLSentence" to perform the construction of the natural language output.

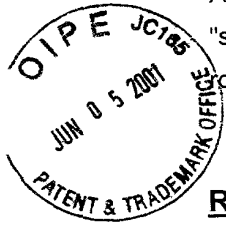
## stdCompareProducts\_ByTopics

This routine receives the names of the products and the topics by which the comparison of said products should be made as input. If no topics were specified as an input, the products will be compared by all available topics. The routine also has a direct access to the memory where the information about said products' topics and values was previously stored by "CompareProducts".

The first attempt, is to find and group similar products. This is accomplished by a call to a function named **"stdFindSimilarProducts"**. The function searches for identical topic-value pairs in the **"cmp\_data(Topic, Value, ProductName)"** entries and gathers the names of the products that have said identical entries into **"similar\_prd(Topic, Value, SimilarProducts)"** structure. The corresponding "cmp\_data" entries are removed from the memory since they are no longer needed. When there are no more **"cmp\_data"** entries to scan , this routine will start scanning the **"similar\_prd"** entries, in order to group all the topic-value pairs by which a given list of products is identical. The result is stored in memory entries of a form **"prd\_similar\_by(TopicValuePairs, ListOfProducts)"**.

Since products may be similar by few topics but different by others and also due to the reason that products may have more than one value associated with a certain topic, there's a need for an additional analysis in order to find these additional topics that differentiate said partially similar products. To perform this task, a procedure named **"stdFindAdditionalInfo"** is called. This routine reads the memory entries **"prd\_similar\_by(TopicValuePairs, ListOfProducts)"** and attempts to find additional **"cmp\_data(Topic, Value, ProductName)"** entries for each product in ListOfProducts. These additional entries, if present, are removed and the information is stored in **"additional\_info(Topic, Value, Product, ListOfProducts)"** where ListOfProducts is the original products list associated with Product in

The last step in performing the comparison between products is to find the distinct products, i.e. - products that can not be grouped under any topic. A good example for this is trying to compare a blue, large and square item with a red, small and round item (these items are not identical by any topic). For this purpose, **"stdCompareProducts\_ByTopics"** will call the routine **"stdFindDistinctProducts"**. The task to be performed by this routine is rather simple - all the information still present in **"cmp\_data"** entries is to be copied to **"prd\_distinct\_by(TopicValuePairs, NameOfProduct)"** entries in the memory while removing the respective **"cmp\_data"** entries.



At this point everything is ready for the construction of the natural language output and "stdCompareProducts\_ByTopics" exits. It is now up to "**CompareProducts**" to decide which routine is to be executed to perform said output, based on the comparison type.

## Relational Compare

### Data entry

Hence is described a module to perform a product comparison with a so called "relational" technique, i.e operate with statements of the form "bigger than..", "rather sweet", "a bit smaller than.." when comparing products.

Since the above statements are known in the art as "fuzzy statements", a fuzzy-logic based implementation of said comparison would be appropriate.

Several types of data need to be defined. The first data entries are the topics by which the comparison is to be made. The definitions of the topics should include the **range** of values that can be associated with the topic (range(MinimumValue, MaximumValue)) , a word to designate the maximal value that can be associated with a product (max\_word(MaximalValuePossible)), a word to designate the minimal value that can be associated with a product (min\_word(MinimalValuePossible)), a word to state a relation of being close to the maximal value (grt\_word(GrtWord)) and a word to state a relation of being close to the minimal value (sml\_word(GrtWord)).

Several words to designate values somewhere inside the range are also allowed. These are done through "mid\_word(Index, MidWord)", Index being used for logical positioning the MidWord inside the range - 1 is the closest to the minimal value, 2 - is a bit farther than the previous and so on.

The following example sheds more light on this issue. Let's examine the topic "taste" assuming the range is "sweet" to "dry" (very reasonable values when discussing wines, for example). When comparing two products we may say "this wine is **sweeter** than the other wine" or "this wine is **drier** than the other wine". In a similar form, a given sort of wine can be "the **sweetest**" or "the **driest**" we have. Another type of wine can also be "semi-dry". Hence, the words "sweeter" and "drier" serve as "sml\_word" and "grt\_word", "semi-dry" is our "mid\_word", and "sweetest" and "driest" are taken as "min\_word" and "max\_word".

However, this is not quite enough to make the comparison sales-oriented. If we want to compare two pairs of speakers, for instance, and use the topic "sound distortions" than: 1) the range is "very low" to "very high" 2) "max\_word" would be "highest" and "min\_word" would be "lowest" and 3) "sml\_word" would be "lower" and "grt\_word" would be "higher". But how can we represent the fact that the lower the sound distortions are the better the speaker is? For this purpose, the "better edge" of the range needs to be defined. The representation would be

0982961-060501

"better\_edge(CharacterThatRepresentsTheBetterEdge) when '+' is the right part of the edge, '-' is the left part of the edge and 'n' symbolizes that there is no "better edge", i.e. any value is neutral.



The second data type represents the connection between said topics and the products. This connection is represented by "**fuzzy\_statement(TopicName,Grade,ProductName)**". That means, that each product is assigned a **numeric value** of the topic, said value being an integral number between 1 and 100. This value positions the product in the range of the topic, so that if we are discussing a certain type of wine, that has the value 20 for the topic "taste" as described above, we will be able to say that the wine is **rather sweet**.

The third data entry describes a prepared stack of words to show amount of difference between products. The representation of this data reads as

**diff\_scale(MinimalNumberOfScale,MaximalNumberofScale,AWordToRepresentTheScale)** where MinimalNumberOfScale and MaximalNumberofScale are numbers and AWordToRepresentTheScale is a string. A further description of this special data entry and where it is used will be given further on.

### Algorithms

#### relCompareProducts\_ByTopics

This routine's input is the names of the products and the topics by which the comparison of said products should be made. If no topics were specified as an input, the products will be compared by all available topics. "RelCompareProducts\_ByTopics" will launch a sub-routine named "relReadDataFile" to load the data entry into operable memory. "RelReadDataFile" will load the "fuzzy\_statement"s where said product names appear. Then, by scanning said "fuzzy\_statement"s "RelReadDataFile" will retrieve the name of the topics mentioned in those statements and load the respective topics with all the data described above. All "diff\_scales" are loaded into the memory as well, since there is no way to tell at this point when they will be needed.

The first operation done at this point, should be scanning the "fuzzy\_statement"s and draw conclusions about the relations between them. The data entries are scanned one by one in the following form: first, the name of the topic is obtained from the first "fuzzy\_statement". Then, "**relCompareProducts\_ByTopics**" calls a procedure named "**relScanStatements**" to find all "fuzzy\_statement"s that contain the same topic. After "**relScanStatements**" finds all "fuzzy\_statement"s that fulfill said condition, it will calculate the highest and lowest numerical values of all entries found. "**RelScanStatements**" returns a list of all products associated with the input topic, along with the respective numerical values, a numerical range of these values (from lowest to highest), the names of the products associated with the highest value and the names of the products associated with the lowest value.

09629961-060601  
T05090-T9652960



After receiving this information, "**relCompareProducts\_ByTopics**" will pass it along with said topic to a sub-routine named "**relDecideOnRelation**" in order to determine which word(s) to use to describe how said products compare with each other.

"**RelDecideOnRelation**" will first construct a scale to decide how the input numerical values relate to the global range for the topic. It will refer to the topic description available in the operative memory and extract a) the range associated with the topic, b) the literal representation of the range (words like - sweet, dry, big, small, etc.) and c) the "mid\_word"s associated with the topic. Then, it will perform the following arithmetical operations: first, it will subtract the minimal value of the range from the maximal value of the range, to receive the amount of units that's inside the range. Second, it will calculate the amount of "mid\_words", add 2 to the result and divide the number of units received previously, by this sum. The result represents the amount of units between various significant values of the range (minimum and the next "mid\_word", first "mid\_word" and second "mid\_word", etc.) and saved in the memory as "**scale\_step(Number)**" for future use.

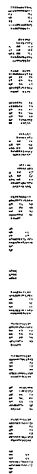
The next and final step is to start adding this number subsequently - from the minimal value of the range to the maximal value of the range, receiving numerical representation of said values as well as numerical representations of "mid\_words". The result is saved in the memory in form of : "**scale(MinimumOfScale, MaximumOfScale, NameOfScale)**."

Here is an example to explain this further: let's assume the topic is "size" with a range of 1 to 100. The minimal value is represented with the word "small", the maximal value - with the word "big". One "mid\_word" is defined - "medium". So, the first step is to subtract 1 from 100. The result is 99. The following step is to calculate the amount of "mid\_words" :  $1 + 2$  equals 3.  $99$  divided by 3 is 33. The last step is to start adding this number subsequently to cover the range - so  $1 + 33$  is 34,  $34 + 33$  is 77 and  $77 + 33$  is 100. Thus we get: **scale(1, 34, "small")** (everything with a value of between 1 and 34 is considered "small"), **scale(34,77,"medium")** (everything with a value of between 34 and 77 is considered "medium") and **scale(77, 100, "big")** (everything with a value of between 1 and 34 is "big").

After that was done, "**RelDecideOnRelation**" will examine the products given as input. If only one product is specified, the corresponding numeric value will be matched against the "scale" entries. In order to fit in one of the scales, the value is to be bigger or equal to the minimal value of the scale and smaller than the maximal value of the scale in the same time. Once the name of the scale the product belongs to is identified, "**RelDecideOnRelation**" will save the result in a data entry of a form

**"prd\_rel\_info(NameOfScale,NameOfProduct,NameOfOriginalAttribute)"**. All information regarding "scale"s and "scale\_step"s is removed from the memory and "**RelDecideOnRelation**" exits.

A different case is when more than one product is passed to "**RelDecideOnRelation**". If that is the situation, several tasks are to be made. First, "**RelDecideOnRelation**" subtracts the minimal value from the maximal value found among these products and compares this result



If the "better\_edge" is '+' (the right side of the relation), "RelDecideOnRelation" will take the "grt\_word" out of the topic description. If it's '-' - the "sml\_word" will be used. The word received in the previous step is appended to the relation word used and the data is stored in the following form: **"prd\_rel\_info(ProductsInTheLeftSideOfRelation, GeneratedWord, ProductsInRightSideOfRelation, EntireListOfProducts, NameOfCategory, NameOfOriginalAttribute)"**.

## Sentence generation

These templates are several of (but not limited to) the following: a) **"simple"** - for cases in



which "a product has a Value Topic" (e.g. - a product has a blue color, a round shape, a big size...), b) **"is"** - for cases in which "a product is Value" (e.g. - a product is blue, big, round), c) **"is\_noun"** - for cases in which "a product is a Value" (e.g. - a product is a shirt, a computer, an apple...), d) **"of"** - for cases in which "a product is of a Value Topic" (e.g. - a product is of a Big Cheese kind, an Expensive look...), e) **"adj"** - for cases in which "a Value product" is a correct english sentence (e.g. - a blue product, a big product, a fast product...), f) **"tv\_adj"** - for cases in which "a Value Topic product" is a correct english sentence (e.g. - a big screen product, a high resolution product, a top quality product...), g) **"suitable\_top\_val"** - for cases in which "a product is suitable for Topic Value" (e.g. - a product is suitable for playing alone), h) **"suitable\_val\_top"** - for cases in which "a product is suitable for Value Topic" (e.g. - a product is suitable for a big room, a blue wallpaper, KX-456 cellular phone model), i) **"other\_t"** - everything other than described, provided the topic is included - (e.g. - "a product is used for cooking", "a product has a center piece"), j) **other\_v** - everything other than described, provided the value is included - (e.g. - "a product is not blue", "a product is made in USA"), k) **other\_tv** - everything other than described, provided the topic and the value are included, in the same order (topic, than value) - (e.g. - "a product is used for cooking breakfast", "a product has a center piece made of stone"), l) **other\_vt** - everything other than described, provided the value and topic are included, topic follows value - (e.g. - "a product is suitable for children under 12 years of age")

These may be defined both per topic or per each value of the topic separately and they reside in a topic's description as a set of `"lang_template(TemplateName, ValueToWhichTheTemplateIsBound, ListOfComplementaryStringsForTheTemplate)`, where name - is the name of the template as described above, `ListOfComplementaryStringsForTheTemplate` contains the two complementary strings for `"other_XXXX"` templates (for `"other_t"` and `"other_v"` these are the two strings before and after the topic or value and for `"other_tv"` and `"other_vt"` these are the string before the topic or value and the string between the topic and value. For the rest of templates this list is empty). `ValueToWhichTheTemplateIsBound` is the name of the topic's value for which the current template is to be used. If `ValueToWhichTheTemplateIsBound` is an empty string, the template will be used to describe the relation between the topic and any of it's values. Every topic and every value of the topic may have unlimited number of templates associated with it.

Furthermore, an additional file should be available, to serve the purpose of a library. This library is to contain some prepared texts in which the program will place the information related to the comparison of the products ( names of the products, topics, values, relational compare information, etc. ). There are 2 types of sentences:

**"compare\_template(NameOfTemplate,Index,String)"** where `NameOfTemplate` is the name of the **comparison template** type, `Index` - an assigned counter used for picking a random text

09039961065001



of said type and String is the text string that contains variables to be replaced by the program. The second type of text is "tv\_template(NameOfLanguageTemplate,Index,String)" - to unify topics and values. NameOfLanguageTemplate is one of the language templates' names described above, plus the keyword "\_mult" or "\_singl" to suit the text for a case when several products are described (and one must say "first product and second product **are** blue, **are** suitable for..., **are** bigger than, **have** a big size") as well as for a case when only one product is described (and one must say "product **is** blue, **is** suitable for, **is** bigger than, **has** a round shape"). The variables Index and String serve the same purposes as those described in "compare\_template" above.

### Algorithms


Two major unit used to construct the output are a routine named "**cmp\_SubstVars**" and another routine named "**update\_compare\_text**". "Cmp\_SubstVars" receives a string where various variables are present and replaces the variables with the relevant content. Whereas the content is to be prepared in advance by the calling routine, **cmp\_SubstVars** identifies the variables in the input string provided that the variables are capitalized and enclosed in brackets followed by the % sign, as in : <%VARIABLE\_NAME%>. The replacement is done using a simple search of the substrings "<%" and "%>" and looking up the variable enclosed in a look up table. The table contains the names of the accepted variables and pointers to the memory where the content to substitute said variables resides.

"Update\_Compare\_Text" receives a string previously processed by "cmp\_SubstVars", the list of products in that string and another string that represents the type of information residing in the previous string (could be either "similar products", "additional info for products", "relational comparison info" or "distinct product"). If the string passed to it is of a "relational compare" type, this routine also receives the names of products on the left and right sides of the relation in separate relation.

First time running, when there's no previous text to append the string to, "update\_compare\_text" will measure the length of the string and write the string to the memory as "prev\_cmp\_text(StrLength, StringType, ProductsList, LeftList, RightList, TheString)". In subsequent calls, this routine will read the "prev\_cmp\_text" from the memory and perform the following steps: first, it will measure the length of the new string passed as input. Then, it will decide how to unify the current string and the previous string, according to their mutual length. If it is smaller than 110, the two strings will be unified to one sentence. Otherwise, each string will remain an autonomous sentence. "Update\_Compare\_Text" will proceed to examine the types of the two strings in order to decide which phrase is to come between the two lines to create a grammatically correct and still human sentence. For this purpose, "update\_compare\_text" accesses it's own internal look up table that consists of "cmp\_txt\_type\_convert(PrevType,CurrType,Phrase)" entries. For example, if PrevType = "similar products" and CurType = "relational comparison info" then one of the possible

096396103501





<p>                     1. <i>Staphylococcus aureus</i> (10<sup>8</sup> CFU/ml)                      2. <i>Staphylococcus aureus</i> (10<sup>6</sup> CFU/ml)                      3. <i>Staphylococcus aureus</i> (10<sup>4</sup> CFU/ml)                      4. <i>Staphylococcus aureus</i> (10<sup>2</sup> CFU/ml)                      5. <i>Staphylococcus aureus</i> (10<sup>0</sup> CFU/ml)                      6. <i>Staphylococcus aureus</i> (10<sup>-2</sup> CFU/ml)                      7. <i>Staphylococcus aureus</i> (10<sup>-4</sup> CFU/ml)                      8. <i>Staphylococcus aureus</i> (10<sup>-6</sup> CFU/ml)                      9. <i>Staphylococcus aureus</i> (10<sup>-8</sup> CFU/ml)                      10. <i>Staphylococcus aureus</i> (10<sup>-10</sup> CFU/ml)                 </p>		<p>                     11. <i>Staphylococcus aureus</i> (10<sup>8</sup> CFU/ml)                      12. <i>Staphylococcus aureus</i> (10<sup>6</sup> CFU/ml)                      13. <i>Staphylococcus aureus</i> (10<sup>4</sup> CFU/ml)                      14. <i>Staphylococcus aureus</i> (10<sup>2</sup> CFU/ml)                      15. <i>Staphylococcus aureus</i> (10<sup>0</sup> CFU/ml)                      16. <i>Staphylococcus aureus</i> (10<sup>-2</sup> CFU/ml)                      17. <i>Staphylococcus aureus</i> (10<sup>-4</sup> CFU/ml)                      18. <i>Staphylococcus aureus</i> (10<sup>-6</sup> CFU/ml)                      19. <i>Staphylococcus aureus</i> (10<sup>-8</sup> CFU/ml)                      20. <i>Staphylococcus aureus</i> (10<sup>-10</sup> CFU/ml)                 </p>
1	11	
2	12	
3	13	
4	14	
5	15	
6	16	
7	17	
8	18	
9	19	
10	20	

The first thing "**construct\_compare\_text**" does is reading the library file described. After this data has been loaded, "**construct\_compare\_text**" will start assembling the natural language output. The beginning of the output is the similar products information (located either in "prd\_similar\_by" entries or in "prd\_rel\_info" entries if only relational compare has been used ). The first entry of this type is fetched and the following operations are performed: first, "**construct\_compare\_text**" will build a line representing the similarity between products. For this purpose "**construct\_compare\_text**" will launch a routine named "**construct\_similar\_products\_text**". This routine, given the names of the similar products and the topic-value pairs by which said products are similar will convert the list of the products to a string by separating the list of products to one product plus the remainder of the list, concating each product name in the remainder of the list with ", " and concating the result with the next product name subsequentially until there are no more elements in the list. The outcome string is then appended to the first product name by using the following formula:

A circular black ink stamp from the Office of Intellectual Property (OIP). The text "OIP" is at the top, "JUN 05 2001" is in the center, and "PATENT &amp; TRADEMARK OFFICE" is at the bottom.

PATENT & TRADEMARK OFFICE

Variable	Mean	SD	Min	Max
Age	35.2	12.5	18	65
Gender	Male	15.8	0	30
Marital status	Married	22.5	0	40
Education	High school	18.2	0	35
Occupation	Unemployed	12.5	0	25
Income	\$15,000	\$10,000	\$0	\$40,000
Health status	Good	15.2	0	30
Smoking	Non-smoker	18.5	0	35
Alcohol	Non-drinker	12.8	0	25
Exercise	Regular	10.5	0	20
Stress	Low	12.2	0	25
Sleep	7-8 hours	15.5	0	30
Diet	Healthy	18.8	0	35
Family size	2-3 children	15.2	0	30
Home ownership	Own	12.5	0	25
Neighborhood	Suburban	18.2	0	35
Transportation	Car	15.5	0	30
Insurance	Health	12.8	0	25
Religion	Christian	18.5	0	35
Political	Democrat	15.2	0	30
Volunteer	Yes	10.5	0	20
Charitable	Donor	12.2	0	25
Community	Active	15.5	0	30
Neighborhood	Safe	18.8	0	35
Transportation	Public	15.2	0	30
Insurance	Life	12.5	0	25
Religion	Muslim	18.2	0	35
Political	Republican	15.5	0	30
Volunteer	No	10.5	0	20
Charitable	No	12.2	0	25
Community	Passive	15.5	0	30
Neighborhood	Unsafe	18.8	0	35
Transportation	Private	15.2	0	30
Insurance	Auto	12.5	0	25
Religion	Jewish	18.2	0	35
Political	Independent	15.5	0	30
Volunteer	Occasional	10.5	0	20
Charitable	Small	12.2	0	25
Community	Neutral	15.5	0	30
Neighborhood	Neutral	18.8	0	35
Transportation	Shared	15.2	0	30
Insurance	Health	12.5	0	25
Religion	Hindu	18.2	0	35
Political	Democrat	15.5	0	30
Volunteer	Regular	10.5	0	20
Charitable	Large	12.2	0	25
Community	Active	15.5	0	30
Neighborhood	Safe	18.8	0	35
Transportation	Public	15.2	0	30
Insurance	Life	12.5	0	25
Religion	Buddhist	18.2	0	35
Political	Republican	15.5	0	30
Volunteer	No	10.5	0	20
Charitable	No	12.2	0	25
Community	Passive	15.5	0	30
Neighborhood	Unsafe	18.8	0	35
Transportation	Private	15.2	0	30
Insurance	Auto	12.5	0	25
Religion	Sikh	18.2	0	35
Political	Independent	15.5	0	30
Volunteer	Occasional	10.5	0	20
Charitable	Small	12.2	0	25
Community	Neutral	15.5	0	30
Neighborhood	Neutral	18.8	0	35
Transportation	Shared	15.2	0	30
Insurance	Health	12.5	0	25
Religion	Hindu	18.2	0	35
Political	Democrat	15.5	0	30
Volunteer	Regular	10.5	0	20
Charitable	Large	12.2	0	25
Community	Active	15.5	0	30
Neighborhood	Safe	18.8	0	35
Transportation	Public	15.2	0	30
Insurance	Life	12.5	0	25
Religion	Buddhist	18.2	0	35
Political	Republican	15.5	0	30
Volunteer	No	10.5	0	20
Charitable	No	12.2	0	25
Community	Passive	15.5	0	30
Neighborhood	Unsafe	18.8	0	35
Transportation	Private	15.2	0	30
Insurance	Auto	12.5	0	25
Religion	Sikh	18.2	0	35
Political	Independent	15.5	0	30
Volunteer	Occasional	10.5	0	20
Charitable	Small	12.2	0	25</

Note: when only relational compare is used, "**construct\_relational\_compare\_text**" also performs these steps: it unifies EntireListOfProducts to a string by the same method used in "**construct\_similar\_products\_text**", get the proper language templates for NameOfOriginalAttribute, choose a random library text string and call "**cmp SubstVars**" to



create the final line of text. This is done to fulfill the mission that otherwise would be performed by **"construct\_similar\_products\_text"** - that is, showing the similarity of said list of products. That line is also passed to **"update\_compare\_text"** and by this **"construct\_relational\_compare\_text"** finishes its mission and exits.

The next step, is to process the so-called "additional information" for said list of similar products. For this purpose, **"construct\_compare\_text"** will launch the routine **"construct\_add\_info\_products\_text"**. This routine is given the initial list of similar products as an input, so it may process the list of products as follows: find all **"additional\_info"** and **"prd\_rel\_info"** entries in the memory for each product, look if same entries exist for any of the rest products in the list and eventually group the topic-value pairs from both entry types to a string by placing them in the memory, calling **unify\_top\_val**, unify the products list and launch **"cmp\_SubstVars"** followed by **"update\_compare\_text"**. Note that whenever an entry that fulfills said conditions is found it is removed from the memory to avoid finding it again and mentioning the information it represents more than once.

After this was done for each product in the list, exits and we're back to **"construct\_compare\_text"**. At this point it's safe to say we've processed the list from the original **"prd\_similar\_by"** entry and we can now proceed to the next entry of the same type. When there are no more similar products lists (i.e. no more **"prd\_similar\_by"** entries in the memory) **"construct\_compare\_text"** will proceed to construct a natural language representation for the products that have nothing in common with any other products, i.e - the **"prd\_distinct\_by(TopicValuePairs, NameOfProduct)"** and the rest of **"prd\_rel\_info(NameOfScale,NameOfProduct,NameOfAttribute)"** entries.

**"Construct\_compare\_text"** will launch **"construct\_distinct\_text"** to process said entries and generate the natural language output. **"Construct\_distinct\_text"** will group the topic-value pairs present in a given **"prd\_distinct\_by"** entry by a call to **"unify\_top\_val"**, previously asserting to the memory the flag **"\_singl"** (since we are discussing one single product here). After this, **"construct\_distinct\_text"** will call **"update\_compare\_text"** to append the string to the previous text.

This process is repeated until there are no more instances of **"prd\_distinct\_by"** in the memory. After the last entry of that kind had been processed, **"construct\_distinct\_text"** exits.

**"Construct\_compare\_text"** calls **"construct\_output\_text"** to perform the output.

**"Construct\_output\_text"** scans the memory to find each **"cmp\_text\_str"** entry, locate the string in it and remove the entry from the memory. After all strings have been found,

**"Construct\_output\_text"** unifies them by concating each string with **"."** and concating the result with the next string, until there are no more strings left to process. The result is one string that contains all the text. The text is then outputted to the standard input (**"stdin"**). Note this is merely one of many possible implementations of the final output.

00000001 00000001